Almost 90% of the extension Add methods in startup.cs are overloaded, one which doesn’t accept anything. One that accepts Action delegate an input parameter.

services.AddControllersWithViews(o =>

           {

               o.InputFormatters.Insert or Add

           });

Add will go ahead and at the bottom of the collection. Insert will go ahead and add at a certain position.

If Add, the default input formatter will get kicked in. first one that’s matched is used. Hence we use Insert with index position as 0.

services.AddControllersWithViews(o =>

         {

             o.InputFormatters.Insert(0, new CertificateInputFormatter());

         });

Our Custom model binder was working with Value provider (OOTB). Now we want to use Custom Input formatter with OOTB model binder.

We need to get rid of custom model binder from the model.

Custom Input formatter is registered to work only with text so the Body -> Raw -> Text .. otherwise won’t hit.

The below is hit which check if our input formatter is capable of reading the certificate type object.

After canread method it next check the ReadRequestBodyAsync is the one that looks for the system.encoding.

protected override bool CanReadType(Type type)

       {

           if (type == typeof(Certificate))

               return base.CanReadType(type);

           else

               return false;

       }

Even for Orders this would be called but at canread it would fails and default would be called.

Even for Json certificate its called, canread passes but ReadRequestBodyAsync fails.

Filters -> Allows the code to run before or after specific stages.

Classes that contain custom logic.

Request -> Other middleware -> Routing middleware -> Action Selection -> MVC action invocation pipeline.

Types of filters -> Authorization, Resource, Action, Exception, Result.

Below is the Filter pipeline: Unified controller filter allows us to use the same filter on both MVC and web API.

1. Authorization -> rules defined by us. Like if you are not using https, then you can’t invoke the certificate controller. But you can still use the Order controller.

Whether the current request should be allowed or not.

In .net core we have rolebased authorization. Use them for any authorization that are not related to users -> like geographical location.

But if manager … blah types authorization (user based) -> don’t use –(he said).

Authentication Filter have been dropped in Asp.net core. They were very messy.

1. There is a new type of filter -> Resource filter.

Once your authorization filter gives green signal for the request, then this is invoked. What do we use a resource filter for? On the return of the response is ready to be delivered to the client and when we receive the request. If this fails, then we short circuit the pipeline. We use it for ex. For custom caching logic. If we have identified that data can be picked from the cache then we don’t go for model binding .. blah blah and return the response to the user.

1. Model Binding only during the request. We serialize
2. Action filter (only during request) (on this itself we can have pre and post logic) For logging purposes. Eveytime the reuest hits the action filter, then just before the action filter gets executed, then the logging happens (like IP, userId). Filters should be for Asp.net core (does extensive logging) but in .net core not so recommended.
3. Exception Filters -> if the action filter code fails, then handle them here. Called once. Instead use Exception Handling middleware.
4. Result Filter -> Pre logic of the result filter then result execution and then post logic and then it goes back to the resource filter.

Scopes and Order of execution:

Scope -> Action, Controller, Global. In what order (if there places where the authorization has been implemented) would the authorization filter be executed.

Order -> Global -> Controller -> Action

Use order property to control the ordering. Filters with index -1 get executed before Global filter. Order can be 0, -1, +1. If you don’t specify the order then all the filters will have default order 0.

If very first then -1, if last then +1.

If we had 2 global filters, and first one has +1 and second one has +2, then +1 gets executed before +2 always. Use it only if extremely required.

Lets create a custom filter now.

<http://quartzsystems.com/downloads/core3/filters.txt>

Reason -> everytime we return the response, suppose we want to send security related details with the response then we can have these custom filters. (content sniffing policy ….read about it)

Security related headers we want – that’s the objective.

We can do this with post Resource, pre or post Action, pre or post result, (headers can be added in the response). What is custom cache is happening.

If pre of resource then action filter developer can override the filters.

We are creating this - SecurityHeaderFiltersAttribute

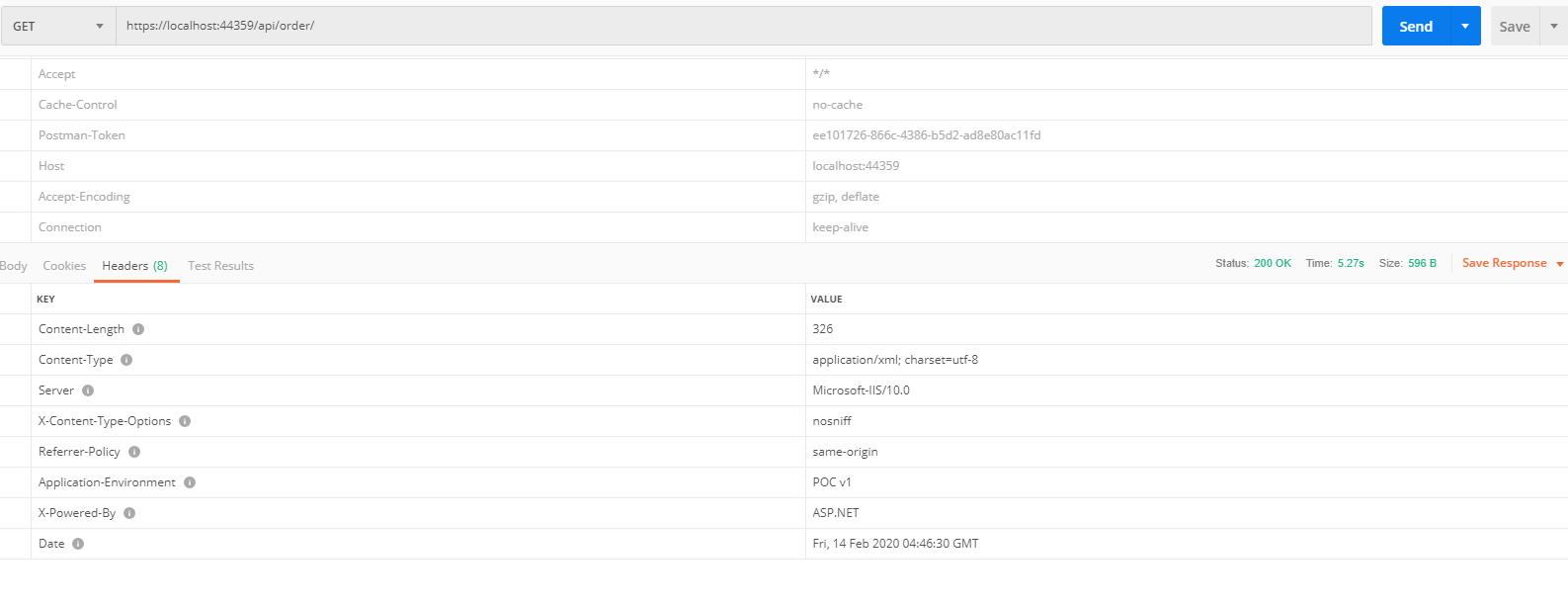
ResultFilterAttribute

Because result, if action filter then change the base class.

It can be applied at any level. Project level then register it in startup.cs.

We doing controller level (add [SecurityHeaderFilters] below namespace.

on making a get request, we see the 3 extra custom filter headers in the response.



How do we check the environment. What is \_environment. Project properties -> debug -> environment variable. IWebHostEnvironment – gives the information about env – server, folder, virtual directory yes or no, and what is the name of the virtual directory. Its automatically registered in IServiceCollections.

Use this object to get env details . OOTB object already registered, to say use it in home controller, inject it in the constructor or other types of injection.

private readonly ILogger<HomeController> \_logger;

       //private readonly ContosoContext \_context;

       private readonly IUnitOfWork \_unitOfWork;

       private readonly IWebHostEnvironment \_environment;

       public HomeController(ILogger<HomeController> logger, IWebHostEnvironment environment)

       {

           \_logger = logger;

           //\_unitOfWork = unitOfWork;

           \_environment = environment;

       }

How do we achieve the same result in the filter? Inject it to the custom filter class.

IWebHostEnvironment \_environment;

       public SecurityHeaderFiltersAttribute (IWebHostEnvironment env)

       {

           \_environment = env;

       }

But now we have 1 parameter constructor, so how do we use it as filter on the Attribute.

Route accepts a fixed string so its okay, but to service filter how do we pass a dependency at the run time to the filter attribute. We are now doing dependency injection in filters.

Filters in Asp.net core can be used instance filter or type activated filters. (an instance is created foe every request or based on its lifetime (if applicable)). **Constructor dependencies are populated by DI.)**

Filters can be registered as services. Singleton – one and only one instance and same instance used everytime for a request.

You should use your filter in conjunction with other filter for making a custom filter and type activated.

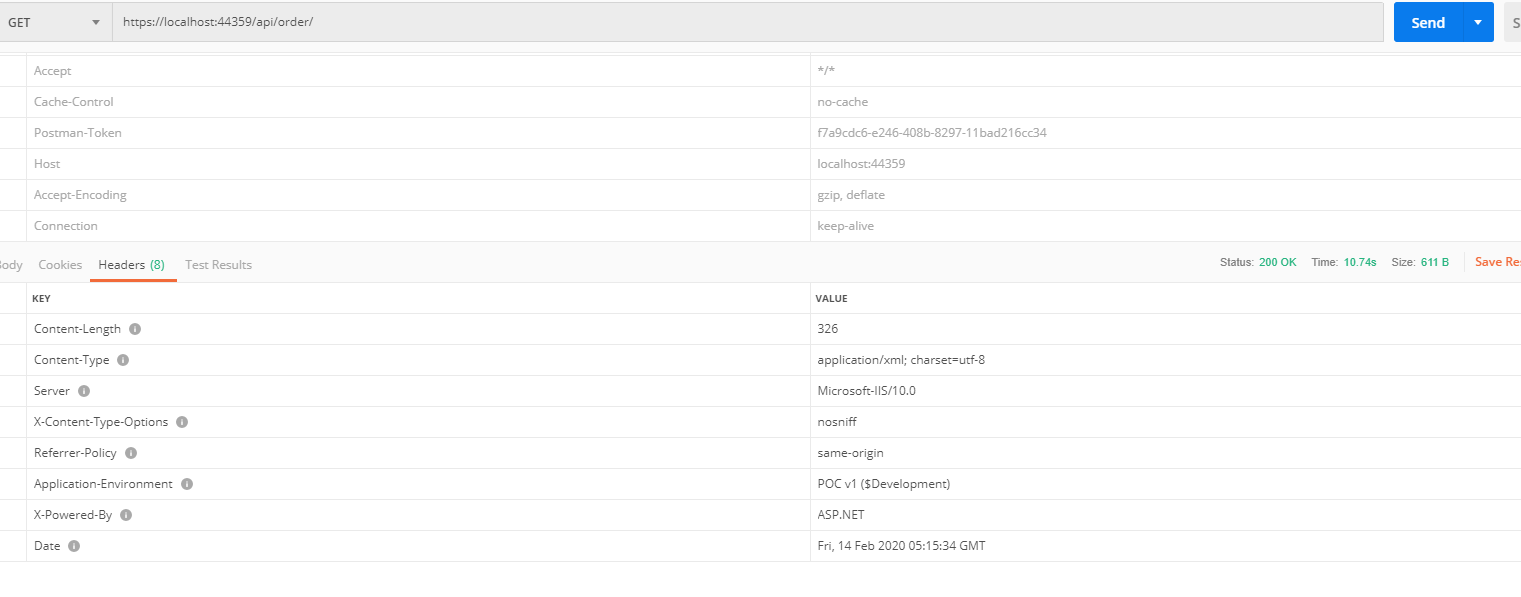
Services Filter and Type filter (ootb) – either of them can be used for this conjunction.

Services Filter – implementation should be registered with DI container.

[TypeFilter(SecurityBlah)]- indirectly using Securityblah – control the lifetime and DI can be used, 2 big advantages.

Nothing needs to be changed on the filter code. [TypeFilter(typeof(SecurityHeaderFiltersAttribute))]

Instead of directly using the [SecurityHeaderFilters] run the code, check env variable – the beauty. New Application environment that gives the env information as well.



Did you know – you can pass the connection string using the environment variable. Which is even more secure. Configuration topic – he will talk in detail.

What if we want 2 separate filters, depending on the env. The filter will get applied, instead of the if condition checking the development env.

Different implementation can be created in different classes. Both implementing the same interface.

SecurityHeaderFiltersProductionAttribute implementing ResultFilterAttribute

We no longer need the contructor for injecting the env variable because filters inside code is not dependent on the environment

ISecurityHeaderFilters – Marker interface created which inherits from the ResultFilterAttribute

And in turn this markup interface is being inherited by the 2 custom filters that we have created.

Now we can use the below code in the controller to apply either of the custom filter to the controller. [ServiceFilter(typeof(ISecurityHeaderFilters))]

But now we need to register this service in the startup.cs class using the below code. See we are adding the lifetime as well.

//Filter

          services.AddSingleton<ISecurityHeaderFilters, SecurityHeaderFiltersProductionAttribute>();

But this code would is not very good, during development we need to change the class that implements the interface for dev env. So we go ahead and use something similar that is present in the Configure method. We inject the object of IWebHost in the method ConfigureServices. Now tell how do we just inject another parameter when actual DI work starts in the ConfigureServices. The IserviceCollection doesn’t really get created here in the startup method. It gets created in the Program.cs when the CreateHostBuilder is run and the some default OOTB services are already added to it like IWebHost, IloggerService, So these can be injected to the configureServices.

//Filter

           if (environment.EnvironmentName == "Development")

           {

               services.AddSingleton<ISecurityHeaderFilters, SecurityHeaderFiltersAttribute>();

           }

           else

           {

               services.AddSingleton<ISecurityHeaderFilters, SecurityHeaderFiltersProductionAttribute>();

           }

But now it gives an error:

**System.InvalidOperationException:** 'The ConfigureServices method must either be parameterless or take only one parameter of type IServiceCollection.'

Instead of adding to configureservices, inject it to the constructor of the startup.

So do the below instead: Read : <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/startup?view=aspnetcore-3.1> (Do I tell him that he is wrong?)

public class Startup

{

private readonly IWebHostEnvironment \_env;

public Startup(IConfiguration configuration, IWebHostEnvironment env)

{

Configuration = configuration;

\_env = env;

}

public IConfiguration Configuration { get; }

public void ConfigureServices(IServiceCollection services)

{

if (\_env.IsDevelopment())

{

}

else

{

}

}

}

This works perfectly fine 😊. And told him. good.

Configure method can have multiple parameters.

**Middleware:**

They are very similar to filters.

* Components that are assembled into an app request processing pipeline.
* Can choose to pass the request the next compenent.
* Can perform work before and after the next component.

Whn to use middleware ? Same thing can be done in filter and middleware. Conceptually similar. But subtle difference. When to use which one.

Middleware pipeline like filter pipeline. Executes for each and every request – whether for MVC, API, not an controller (like for cs file, jpg file) in the application.

Filter pipeline is executed only for the requests received for the controller.

Does your custom logic needs to be implemented for every single request -> then use middleware. Fundamental point to govern whether to use it or not.

Every time the request is request, each middleware component statcked in order (in startup.configure) is called.

Control passed from one to another. **Terminating middleware** is the one that doesn’t pass the control to the subsequent component (definitely some business rule didn’t pass).

Every middleware component called twice. During request and response.

Adding the middleware component (Request Delegates)

Run, Use, Map->

Run a single/terminal delegate

Use can be used to chain delegates.

Map conditionally invokes the delegates.

Inline middleware - ????????? google

<http://quartzsystems.com/downloads/core3/middleware.txt>

app.UseStaticFiles(); - in some cases short circuits the pipeline – terminating middleware.

Move all code currently to a different method.

//for the time being comment this default middleware

            //RegisterDefaultMiddlewares(app, env); <- all the existing middleware.

            RegisterDefaultMiddlewares(app);

Use(context, next) -> context is HTTPcontext and next is the pointer to the next middleware component. What if no next, then the rest of the component will never get called. Asp.net core doesn’t check whether next set is getting called or now. This is your logic.

Difference between run and use. -> use method delegate has 2 params, run has 1 (no pointer to the next component) Well you wouldn’t be able to call the next component. You utilize the Run method when you want to create a terminating method. By convention in USE you should always pass the control to next comp. but with Run, the meiddleware short circuits.

In case of unhandled exception, the middleware short circuits and the exception gets bubbled up (if not handled there itself) and the UseExceptionHandler on top will handle the exception in any component. This is why it is on the top of the stack. Use try catch handling other middleware components. Use try catch for your logic on your middleware only. (wrong below code)

try

                {

                    await next.Invoke();

                }

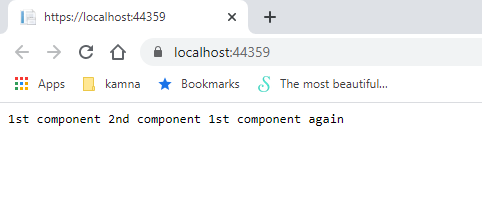
                catch

                {

                    throw;

                }

The below is the output now



public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

     {

         //for the time being comment this default middleware

         //RegisterDefaultMiddlewares(app, env);

         RegisterDefaultMiddlewares(app);

     }

     private void RegisterDefaultMiddlewares(IApplicationBuilder app)

     {

         //these use and run are using action delegate. (there is something called func delegate as well

         app.Use(async (context, next) =>

         {

             //do some work

             await context.Response.WriteAsync("1st component ");

             await next.Invoke();

             //do some more work

             await context.Response.WriteAsync("1st component again ");

         });

         app.Run(async (context) => { await context.Response.WriteAsync("2nd component "); });

     }

     private void RegisterDefaultMiddlewares(IApplicationBuilder app, IWebHostEnvironment env)

     {

         if (env.IsDevelopment())

         {

             app.UseDeveloperExceptionPage();

         }

         else

         {

             app.UseExceptionHandler("/Home/Error");

             // The default HSTS value is 30 days. You may want to change this for production scenarios, see https://aka.ms/aspnetcore-hsts.

             app.UseHsts();

         }

         app.UseHttpsRedirection();

         app.UseStaticFiles();

         app.UseRouting();

         app.UseAuthorization();

         app.UseEndpoints(endpoints =>

         {

             endpoints.MapControllerRoute(

                 name: "default",

                 pattern: "{controller=Home}/{action=Index}/{id?}");

         });

         app.UseSwagger().UseSwaggerUI(o => o.SwaggerEndpoint("/swagger/v1/swagger.json", "Heading I can customize"));

     }

Sometimes when we want to conditionally call pipeline component

3 variants of Map -> Map MapWhen, UseWhen.

Map and MapWhen (have some kind of predicate)

private void RegisterDefaultMiddlewares(IApplicationBuilder app)

        {

            app.Map("/en", a =>

            {

                a.Use(async (context, next) =>

                {

                    await context.Response.WriteAsync("en component ");

                    await next.Invoke();

                });

            });

            //these use and run are using action delegate. (there is something called func delegate as well

            app.Use(async (context, next) =>

            {

                //do some work

                await context.Response.WriteAsync("1st component ");

                await next.Invoke();

                //do some more work

                await context.Response.WriteAsync("1st component again ");

            });

            app.Run(async (context) => { await context.Response.WriteAsync("2nd component "); });

        }

This map middle ware is associated with the condition. With map we look at the concept on request path. If reuquest has /en then the middleware components get added.

See this map has created a branched pipeline. Outside map is separate branch (default)

Inside is another branch.

Right now my out is same as previous. Lets add /en but yet to see output.

app.MapWhen(context => context.Request.Path.Value.EndsWith("/en"), SetCultureEnBranch);

its calling the method SetCultureEnBranch.

Samething here as well. Branching of the pipeline happens. await next.Invoke(); is good practice. Tomorrow new component may be added to the pipeline.

UseWhen is like MapWhen but rejoins the branch to the main pipeline.

app.UseWhen(context => context.Request.Path.Value.EndsWith("/en"), SetCultureEnBranch);

1st component 2nd component 1st component again <- again gets printed in addition to the SetCulture method.

Now we create a custom middleware component.

SecurityHeadersMiddleware – It doesn’t inherit from anything.

InvokeAsync

Has the custom logic.

await \_next(context); this line is an extra on the custom filter. Because we don’t want this custom middleware as a terminating middleware.

Should have reuestDelegate parameter in the constructor.

Public InvokeAsync with Httpcontext as the input param.

Invoke the provided Request

Create an Extension method to expose custom middleware component

Use usemiddleware to invoke the

MiddlewareExtensions.cs is the one that is used to invoke.

app.UseEndpoints(endpoints =>

            {

                endpoints.MapControllerRoute(

                    name: "default",

                    pattern: "{controller=Home}/{action=Index}/{id?}");

            });

This middleware when sees that the request to the controller then this acts as a terminating middleware.

That’s why swagger is added after that because swagger comes into picture when above is used.

So we put ours above useendpoints.

What is the difference in the functionality that we are achieving between filters and meddlewares ?

When we have some logic that is applicable only to controllers and action, then we should do it via filters.

Authentication should be done by the middleware.

app.UseStaticFiles(); if it finds the static file, it executes and it acts as a terminating middleware.

/css/bootstrap.css

Use cutom on top of the above line so our code works for both controllers and the static files. Headers get added.

Middleware is right now by default singleton. But now lifeline issue so it can’t use DBContext which is a scoped.

Factory based Activation helps us have the middleware registered as scopes/transient.